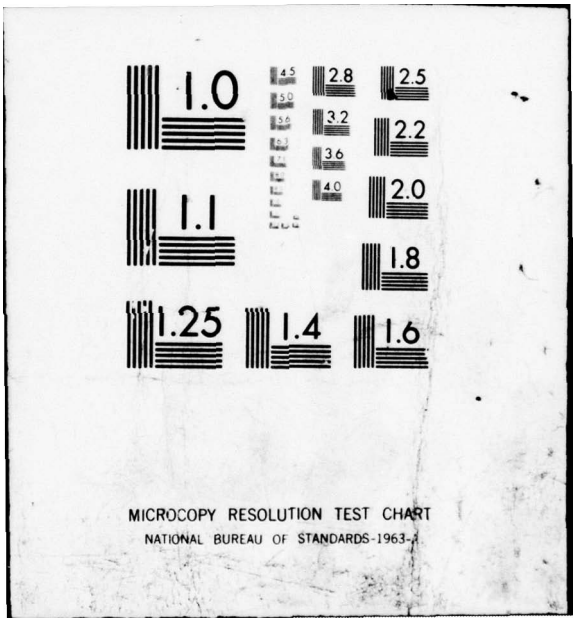
AD-A065 196 MCDONNELL DOUGLAS ASTRONAUTICS CO HUNTINGTON BEACH CALIF F/G 9/2
METRICS OF SOFTWARE QUALITY,(U)
JUL 78 Z JELINSKI, P B MORANDA F49620-77-C-0099
UNCLASSIFIED MDC-G7517 AFOSR-TR-79-0128 NL

| OF |
AD
A065196

END
DATE
FILMED
4-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

MCDONNELL DOUGLAS ASTRONAUTICS COMPANY

MCDONNELL DOUGLAS CORPORATION

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

Approved for public release;
distribution unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AFOSR-TR-79-0128	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: Metrics of Software Quality	5. TYPE OF REPORT & PERIOD COVERED: Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Z. Jelinski and P.B. Moranda	8. CONTRACT OR GRANT NUMBER(s): F49620-77-C-0099
9. PERFORMING ORGANIZATION NAME AND ADDRESS: McDonnell Douglas Astronautics Company, 5301 Bolsa Avenue, Huntington Beach, California 92647	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling AFB, wAshington, DC 20332	12. REPORT DATE: July 1978
	13. NUMBER OF PAGES: 57
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Software metrics
Test tools

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report covers the period from 1 June 1977, to 31 May 1978, during which the first of three phases was completed. This phase concentrated on identifying existing strategies for testing and composing new ones for use in the subsequent phases in which software metrics, related to testing, will be developed and tested against a variety of computer programs.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

389310 R.L.H.

20. Abstract continued.

The literature review was in the main disappointing. The software metrics identified numbered over 50 and of these about 5 appear capable of critical review and ultimate use. The literature on testing produced one technique, due to W. Miller and D. L. Spooner, which appears to lie along the direction of the proposed research. Other techniques were reviewed for content.

The aim of the research is to produce an automatic testing tool which will exhaustively test all tracks through a program, where a track is a time integrated shadow of the execution sequence - where multiple passes through a program segment are recorded as a single pass and the order of usage of the segments is not relevant.

A framework for integrating the Miller and Spooner technique into the automatic mode has been established.

LEVEL II

3

14 Report MDC-G7517

12 598

18 AFOSR

19 TR-79-0128

6 METRICS OF SOFTWARE QUALITY,

10 Z. Jelinski
P. B. Moranda

16 2304

17 A2

McDonnell Douglas Astronautics Company
5301 Bolsa Avenue
Huntington Beach, California 92647

11 July 1978

9 Annual Report for Period 1 June 1977 - 31 May 1978

Contract ~~F49620-77-0099~~ F49620-77-C-0099

15

Prepared for:
Air Force Office of Scientific Research
Bolling Air Force Base, D.C.
D.C. 20332

Attention: Lt. Col. George W. McKemie, Contract Monitor
Directorate of Mathematical and Information Sciences

389 310 RLH

SUMMARY

This report covers the period from 1 June 1977, to 31 May 1978, during which the first of three phases were completed. This phase concentrated on identifying existing strategies for testing and composing new ones for use in the subsequent phases in which software metrics, related to testing, will be developed and tested against a variety of computer programs.

The literature review was in the main disappointing. The software metrics identified numbered over 50 and of these about 5 appear capable of critical review and ultimate use. The literature on testing produced one technique, due to W. Miller and D.L. Spooner, which appears to lie along the direction of the proposed research. Other techniques were reviewed for content.

The aim of the research is to produce an automatic testing tool which will exhaustively test all tracks through a program, where a track is a time integrated shadow of the execution sequence - where multiple passes through a program segment are recorded as a single pass and the order of usage of the segments is not relevant.

A framework for integrating the Miller and Spooner technique into the automatic mode has been established.

METRICS OF SOFTWARE QUALITY

1. OBJECTIVES AND TASK DESCRIPTIONS

1.1 Work Accomplished

The effort during the contract period consisted of four tasks:

1. Task I - Review contemporary work of researchers in software testing field to postulate testing strategies.

2. Task II - Perform preliminary tests of selected programs to obtain some data on various testing strategies.

3. Task III - Evaluate parameters influencing software quality to suggest appropriate metrics.

4. Task IV - Document as appropriate to facilitate later extensive experiments.

1.2 Additional Work

The goals of the research are to develop a preliminary design and partial implementation of a message entry/computer/display combination for interactive testing and case selection so as to achieve exhaustive testing of arbitrary (FORTRAN) programs. Studies will also he made to assess the practicality of a fully automated version of testing.

This additional work can be subdivided into the following major tasks:

A. Experiment with Software Quality metrics which will include:
   1. Selection of FORTRAN Programs for strategy testing of techniques identified and investigated in Sections 2.1 and 4.2 of this report.

2. Analysis of the programs using appropriate techniques.
3. Composition of the extended version of the Program Testing Translator, integrating into the pre-processor the means of augmenting program code to provide valuations of the program predicates, and values of the artificial program variables providing the data for the search procedures.
4. Determining the level of testedness in each of the selected programs, and developing means of automatically selecting test cases to achieve specified paths through the tested programs.

B. The development of a methodology for developing candidate drivers for untested regions of a program; this consists of specifying starting points, on the input domain, likely to cause execution sequences producing designated predicate valuations.

C. Tailor or expand the testing programs developed and used in A and B above. The necessity and benefits of expanding the procedure to include a larger set of program predicates will be studied.

D. Modify, install and test the Tool on a laboratory computer when the scope and size of the general purpose test tool is established. The use of macros, augmenting an intermediate language, will be investigated on a trial basis.

E. Test the Tool and the methodology using the several identified constructs of the text (connection matrix, status vectors, predicate valuations, and input and output data) through the implementation on a "laboratory" type of computer, such as the Nanodata QM-1.

2. STATUS OF THE RESEARCH EFFORTS

2.1 Reviews

Two different levels of review were made, one is a thorough and complete analytical level, the other for content only.

2.1.1 Review of Contemporary Work

Rather extensive and detailed examinations were made of the literature of the software testing field and of software metrics in general. The following papers were reviewed indepth.

1. TRW Software Reliability Study. TRW Final Report, RADC TR-76-236, August 1976.
2. M. L. Shooman, "Structural Models for Software Reliability Predictions", Proceedings of the 2nd International Conference on Software Engineering, 13-16 Oct 1976, San Francisco, California.
3. H. E. Williams, T. A. James, A. A. Beaureguard, and P. Hilcoff. "Software Reliability Systems: A Raytheon Project History", RADC-TR-77-188, Final Technical Report, June 1977.
4. IBM Federal Systems Division, "Statistical Prediction of Programming Errors" RADC-TR-77-175 Final Technical Report, May 1977.
5. Doty Associates, Inc., "Software Cost Estimation: Vol. 1", RADC-TR-77-220, Final Technical Report, June 1977.
6. J. R. Brown, H. N. Buchanan "The Quantitative Measurement of Software Safety and Reliability" SDP 1776, 24 August 1973.
7. M. Shooman and A. Laemmel "Statistical Theory of Computer Programs - Information Content and Complexity" Digest of Papers Fall COMPCON 77, Washington D.C., 6-9 September 1977.
8. G. J. Schick and R. W. Wolverton "An Analysis of Competing Software Reliability Models" IEEETSE, March 1978; Vol. SE-4, No. 2, (reviewed in this section 3.1.8).
9. G. J. Myers, Software Reliability, Wiley-Interscience, 1976.
10. A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science", ACM Computing Surveys Vol. 10, No. 1, March 1978.

2.1.2 Literature Reviewed for Content

1 Z. Manna. Mathematical Theory of Computation, McGraw-Hill, Inc., New York, 1974.

2 T. Gilb, Software Metrics, Winthrop Publishers, Inc., Cambridge, Mass., 1977.

3 A. Goel. "Bayesian Software Predictions Models," RADC-TR-77-112, March 1977.

4 M. Shooman, "Manpower Deployment Effects on Software Error Models," in RADC-TR-76-143, May 1976.

5 Boeing Computer Services, "Software Data Acquisition," RADC-TR-77-130, April 1977.

6 W. H. Howden, "Methodology for Generation of Program Test Data," IEEE TransComp, Vol. C-24, May 1975.

7 L. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEETSE, SE-2, 1976.

8 S. Gerhart and L. Yelowitz, "Fallibility in Applications of Modern Programming Techniques," IEEETSE Vol. SE-2, No. 3, Sept. 1976.

9 R. F. Serfozo, "Compositions, Inverses, and Thinning of Random Measures," Syracuse University, Dept. of Ind. Eng. and Ops. Research, December 1975.

10 L. Osterweil, "Depth-First Search Techniques and Efficient Methods for Creating Test Paths," Univ. of Colorado Dept. of Comp Sci TR No. CU-CS-077-75, August 1975.

11 W. Miller and D. Spooner, "Automatic Generation of Floating Point Test Data," IEEETSE Vol. SE-2, No. 3, Sept. 1976.

12 G.E.P. Box and K. B. Wilson, "Attainment of Optimum Conditions," J. Royal Stat Soc., Vol. XIII, No. 1, 1951.

2.1.3 Review of Testing Tools and Procedures

Recent articles of a review nature have identified and described a large number of different testing tools. D. J. Reifer (Reference 1) identified 70 different types of tools and briefly discussed each type. C. V. Ramamoorthy and S. F. Ho (Reference 2) discuss, in some detail, 15 different tool types. A review of these different types here would be duplicative. Instead a composite review of the limited number of reports listed in above dealing with the testing process will be presented. Usually the potential deficiencies of the processes or tools are brought out in the description, but not their advantages.

Before the discussion of individual classes of tools is undertaken here, it is well to note that the paper by Goodenough and Gerhart (Reference 3) illuminates many of the heretofore neglected points concerning testing.

Some of the important points they make in this respect are:

A. It is not enough to execute a statement with a particular set of conditions, it must be tested in all combinations of conditions;
B. In the same sense, a path through a loop may have to be taken several times before the conditions for error revelation are met;
C. Missing, but required-for-correctness, components of a program (such as predicates or assignments) clearly cannot be identified by "cover-testing" a program;
D. Generally a program must be examined for what it actually does instead of what the tester is told the program does and, at each point of interface, it must be examined for what it can do;
E. The environment, including the operating system, hardware processor and language, have to be examined.

2.1.3.1 Inside Out Testing

Several different techniques have been employed to develop test cases on the basis of a specified set of valuations or outcomes of the program's predicate. The mathematical expressions employed in the program predicates, are used to develop a set of restrictions on the input data space. Solution of the set of equations then produces a point or set of points that will achieve the path through the program. The difficulty with the procedure is that the set of equations involved often are not tractable, even for cases where only "area" (as distinct from point) solutions are required. This difficulty is, to a degree, alleviated by use of interactive entry of data and display-guided solutions.

2.1.3.2 Symbolic Execution

Instead of operating on numerical (or logical) values for variables in a processing, the program's operations can be carried out on the symbols themselves. This technique was independently proposed by W. E. Howden (Reference 4) at McDonnell Douglas Astronautics Company, B. Elspas, et al. (Reference 5) at Stanford Research Institute, J. C.King of IBM (Reference 6) and Lori A. Clark (Reference 7) of the University of Colorado.

Programs, so exercised, must be augmented so they become capable of symbolic execution of expressions and provide means for selecting specified branches or paths in them. Howden employs a system (DISSECT) processing the program that is to be symbolically executed, along with a list of commands that cause symbolic execution.

The advantages of symbolic execution are clear. In certain cases the printout consists of an explicit formula that is unambiguous to the reader. If the formula is correct, the program is correct for all data and there is no necessity for numerical comparisons or independent checks.

In many cases however, the output is far from clear to any but the most experienced users. There is, for example, sometimes a need to maintain the list of possible antecedents (a suspense file) for a program variable having several different symbols and values assigned to it. Further there is a context-dependency that a given assignment may have, caused, for example, by different encounters

of an assignment during looping. This must be accounted for, and in the case of DISSECT, the context is identified by a number representing the dynamic instruction number (as distinct from the static sequence number associated with a listing). These and more complex problems have been faced by Howden and others and they provide finished products that are proof that such techniques can be used to good effect when the tools are in the hands of experts.

While there are some barriers to the "field" use of such techniques, they do not seem insurmountable and it is probably reasonable to expect that symbolic execution can be of common use.

2.1.3.3 Automated Verification Systems

Several systems instrumenting a given program to permit the tallying of the uses of its instructions, branches, and so forth, are classified as automated verification systems by Reifer (Reference 1). They are usually not automated in the strict sense of the word. Although they require a set of input test data to drive the program, there is no instantaneous feedback to change the data to test new unexercised sections of the program. A complaint on word usage can be also made that these systems do not really verify the tested program, and generally do not even consider the output in respect to its accuracy, or even its relevance.

A McDonnell Douglas Automation Company tool, called PET (for Program Evaluator and Tester) described by L. G. Stucki in a company report (Reference 8) and in the open literature (Reference 9), is typical of this class.

For a given data set, PET reports the usage by instruction and branch, which the execution sequence represents. There are other useful metrics, including the range of value for each of the program variables. Lists of unexercised program components also are printed out.

An augmented version of PET, that formed segments consisting of "dynamically contiguous" program instructions, was used and described in a recent AFOSR-sponsored study (Reference 10). In that study, as with most other applications of PET, the emphasis is on the "coverage" of the tested program. Repeated tests with randomly generated input data were used, and their effects merged to produce a composite (montage) of the testing status of the program. Unexercised segments

were used to find the governing predicate or predicates in the program listing, and so-called constructed cases were then formed. The process was continued as far as deemed possible to establish the testing degree.

This class of program monitors is useful in another way. Frequently exercised portions of a program can be identified by the tallies or counts and the identified regions can be examined to see if improvements can be made in the coding or basic algorithms.

2.1.3.4 Automatic Test Generators

Conceivably any particular segment of a program has some input data that will cause it to be exercised. Since it is possible, as indicated in the section on inside-out testing, to back up from a particular point in the program to the "top," it should be possible to choose a set of inputs that will cause any given segment to be exercised. The technique used amounts to an identification of the program variables that are "active" at the segment, and then to relate these to the input variables. This is illustrated in Section 2.2 where the precise set of relations to the input data are developed explicitly from a particular "straight line" path through the program .

Usually it will not be necessary to develop the precise relations (which, it is noted,is almost the same as symbolic execution) between the program variables and the input, and it is only necessary to identify those inputs affecting the selected program variable. This can be accomplished in an even less elegant way by simply generating random numbers to serve as values for the input variables.

Whatever scheme is used, the automatic test generators provide a basis for economically meaningful testing-to-"completion." The idea is simply to form a "feedback" loop between a cumulative record of the segments previously tested to, what might be called, a scenario generator. The scenario generator would provide a one-at-a-time selection for the untested segments, and the standard test generators could be used to "find" the required data. This will then cause the new scenario update and a new selection. This idea is mentioned again later in connection with the use of "tracks" and the planned automatic case generation process.

2.1.3.5 Domain-Testing Strategies

The point mentioned above, in connection with the possible creation of a truly automatic test tool, brings up the important problem identified earlier, the essential impossibility of producing a particular numerical value by the usual kinds of random number generation. This is not a problem in estimating the asymptotic limits to testing with such numbers, because of the infrequent occurrence of these numbers in the sample. For the development of constructed cases where exhaustive testing can be achieved, it is necessary to specify the set of points in the input space which, after processing, will produce a specified value for a program variable.

Generally speaking, the particular set of points achieving the specified value has relatively small dimensionality (a point in two-space, a line in three-space, etc.) making the problem of testing boundaries important.

E .I. Cohen and L. J. White of Ohio State University (Reference 11), have investigated this and similar problems and developed strategies that will test domains with linear and non-linear boundaries (the latter only in two dimensions at present) in efficient ways. As noted, work of this kind is essential to any ultimately automatic testing scheme.

2.2 Preliminary Tests

2.2.1 Preliminary Technical Discussion

The primary purpose of this section which contains a background for later discussion is to explain the framework in which the testing will occur. Also important is the description of the criteria used in choosing software metrics.

2.2.1.1 Description of Testing Framework

Under AFOSR contract AF 44620-74-C-0008, MDAC developed a model which employs random numbers as input data, and, on the basis of the trial numbers on which "new" logical paths are driven by the input data, estimates the asymptotic, or eventual, level of testing achieved with random numbers. The basic analysis mechanism is the original Jelinski-Moranda model (Reference 12). The measurement used in the model is the number of trials occurring between the discovery of new

logical paths (rather than times between errors which comprised the raw data for the estimation of residual error content in the original application of the model).

There is another relevant use of this same model. If the probability law governing the selection of input data is known, then the coupling of information derived from sampling with universal (a priori) error rate data will permit an estimate of the operational reliability of the program. This procedure, also developed under the same AFOSR contract was reported in Reference 13.

A second model employing program or software input data for analysis, is due to TRW (Reference 3). In essence, this model uses a subdivision of the input data space into equivalence classes, each characterized by the particular logical path exercised by all of its members.

This subdivision was suggested earlier by W. Howden (Reference 4) and also by B. Elspas, et al., (Reference 5). In applications the TRW model has been used in the estimation of software reliability. The estimate is derived by composing the assumed-to-be-known probability that each subdivision is employed, with a sample-derived conditional probability of committing an error when the subdivision is used. The problem in the application of such a model is the difficulty involved in the formation of the subdivisions, confirmed by almost everyone who has attempted to work from a specified logical path to the descriptor of the input data associated with it. Another deficiency occurs when the model is used for estimation because, a permanent program is assumed which does not change to remedy the found errors. The problem of precisely carving out the equivalent classes is a severe barrier to application of such techniques. It is probably better to avoid the problem, as done in the application of random numbers described in Reference 10, or by using techniques like those described by W. Miller and D. Spooner (Reference 14).

The use of random numbers as inputs to a software package has fundamental limitations. For example the occurrence of an input which takes on a zero value is essentially impossible and this input, and others of a similar nature, must be supplied to produce a set of inputs which will achieve such values.

Nevertheless, as shown in earlier work (Reference 10), the fundamental limitation can be numerically estimated for a given program on the basis of the set of logical paths effected as a result of random drivers. It can be said that the number found in this way is an always fair and often an excellent bound on the total number of logical paths which are ever actually exercised.

The work of Miller and Spooner avoid these problems with an elegant substitute: instead of attempting to solve, in the input data space, the set of equations (or inequalities) associated with a specified logical path, they insert a new set of variables, one at each branching point in the program. An objective function of these variables is chosen so that when its functional value is positive, the input data is in the equivalence set associated with the specified logical path. This method employs standard procedures from the field of system optimization, starting with a randomly chosen initial point in the input domain.

For additional background, a review is made of the means of representing the flow graph by a connection matrix. As noted in prior work the matrix is constructed by assigning a 1 or 0 as an entry, according to whether or not there is a connection between the nodes (or segments) corresponding to the associated row and column of the matrix. A simple way of visualizing the problem of exhaustive testing can be posed in matrix format. Since a connection matrix C is a descriptor of potential links between segments, the execution sequence in response to an input data value x_1 (in most applications x_1 is a vector instead of a scalar), can be associated* with a submatrix of C, say S_1. Since C is finite, the problem of exhaustive testing can be framed as follows: for C a given connection matrix find a set $x_1, x_2, \ldots x_m$, so that for the associated submatrices $S_1, S_2, \ldots S_m$

$$C = \bigcup_{i=1}^{m} S_i$$

---

*As discussed in Reference 10, an execution sequence can be mapped to submatrix by ignoring the ordering of its branches. This is valid only because of the definition used here for exhaustive testing.

where $\mathop{B}_{i=1}^{m} S_i$ represents the Boolean union or sum of the S_i. (This essentially defines the nature of exhaustive testing.)

An efficient test would be one in which the number of test points, m, is minimal.

As noted above, essentials of the process involve associating with each decision point (two-way) or predicate within the program, a function which has a non-negative value when the predicate is true, and negative value when it is false. In many cases, such as comparison between program variables by inequalities, the expression in the predicate can serve directly to define the function. If, for example, there is a test $P \leq Q$, then the variable assignment, or function, C=P-Q, can be used. Since the functions are relations among variables, they can be considered to be program variables. By forming variables of this kind at each branch point, the program is augmented in such a way that, in response to an input data set, an execution sequence will take place in which values are given not only to all program variables but also to the augmenting variables, which, as noted, are program variables.

Because the signs (+ or -) of the augmenting variables, set up a unique pattern for any input data, they can be used to define the equivalence classes mentioned above. It is noted again that in the formation of the equivalence classes the ordering of the sequence has been ignored.

In a different mode of usage, the sign of each of the augmenting variables can be specified in advance, and a point (or region) in the input data space causing this pre-specified pattern of signs can be sought. By assignment of any of a number of simple objective functions of the augmenting variables, with properties described subsequently, the problem can be stated as a search problem generally identified with optimization problems. Generally the search is made to find data which will make the objective function positive; it is not necessary to achieve a maximum for the objective function, only that the value of the function be positive. This problem is more simple than the optimization problem.

A technique, due to W. Miller and D. Spooner (Reference 14) is illustrated by their example shown below. Their description of the example has been augmented in several ways.

The problem of the example is one of triangularization of an NxN matrix by Gaussian elimination.

The original code is shown in Figure 1. A combined flowchart and code with predicates and branches identified, is shown in Figure 2. The predicates, shown enclosed in rectangular boxes are attached to the node representing the site of their occurrence. The augmented code employing the functions associated with the predicates, is shown in Figure 3. The input data to the program consists of the nine matrix entries: A(1,1),A(2,1)...,A(3,3).

Formation of the augmented code is accomplished by making a straight line pass through the program under the assumption that the predicates inside the DO loop 1 are all true, and all of the rest are valued false. It is noted that the test results denoted as K=N, are governed by the input assignment to the matrix order, N, here taken in the example as N=3. There is a "false" valuation until K=3. These valuations are implicitly made in the construction of the program into a straight line representation. Similarily the tests, denoted as M=K, are completely determined by the tests in the DO loop 1 and do not explicitly show in the augmented code; they are used to develop the straight line code.

The variable C_1, shown on the first line of Figure 3, is positive if the predicate, ABS (A(2,1))-ABS (A(1,1)), is true; and this condition has been specified as holding, since the predicate is in the DO loop 1. A similar remark applies to C_2 and C_7 in the straight line listing because they are repeats of the same test encountered under new conditions. On the other hand, the two tests shown in Figure 2, denoted as T=0, are taken to be false on each encounter, and the value C_3, C_4, C_5, C_8 and C_9 will all test positive if the false branches are to be taken. (Since it is only required that T be non-zero, the C's could also be chosen to be negative, but the analysis is tailored around positive valuations.)

```
      IP(N) = 1
          DO 6 K = 1,N
                IF (K.EQ.N) GO TO 5
                KP1 = K+1
                M = K
                DO 1 I = KP1,N
                      IF (ABS(A(I,K)).GT.ABS(A(M,K))) M = I
    1           CONTINUE
                IP(K) = M
                IF(M.NE.K) IP(N) = -IP(N)
                T = A(M,K)
                A(M,K) = A(K,K)
                A(K,K) = T
                IF (T.EQ.0.) GO TO 5
                DO 2 I = KP1,N
    2                 A(I,K) = -A(I,K)/T
                DO 4 J = KP1,N
                      T = A(M,J)
                      A(M,J) = A(K,J)
                      A(K,J) = T
                      IF(T.EQ.0.) GO TO 4
                      DO 3 I = KP1,N
    3                       A(I,J) = A(I,J) + A(I,K)*T
    4           CONTINUE
    5           IF (A(K,K).EQ.0.) IP(N) = 0
    6     CONTINUE
          RETURN
          END
```

Figure I. Coding for Example Program

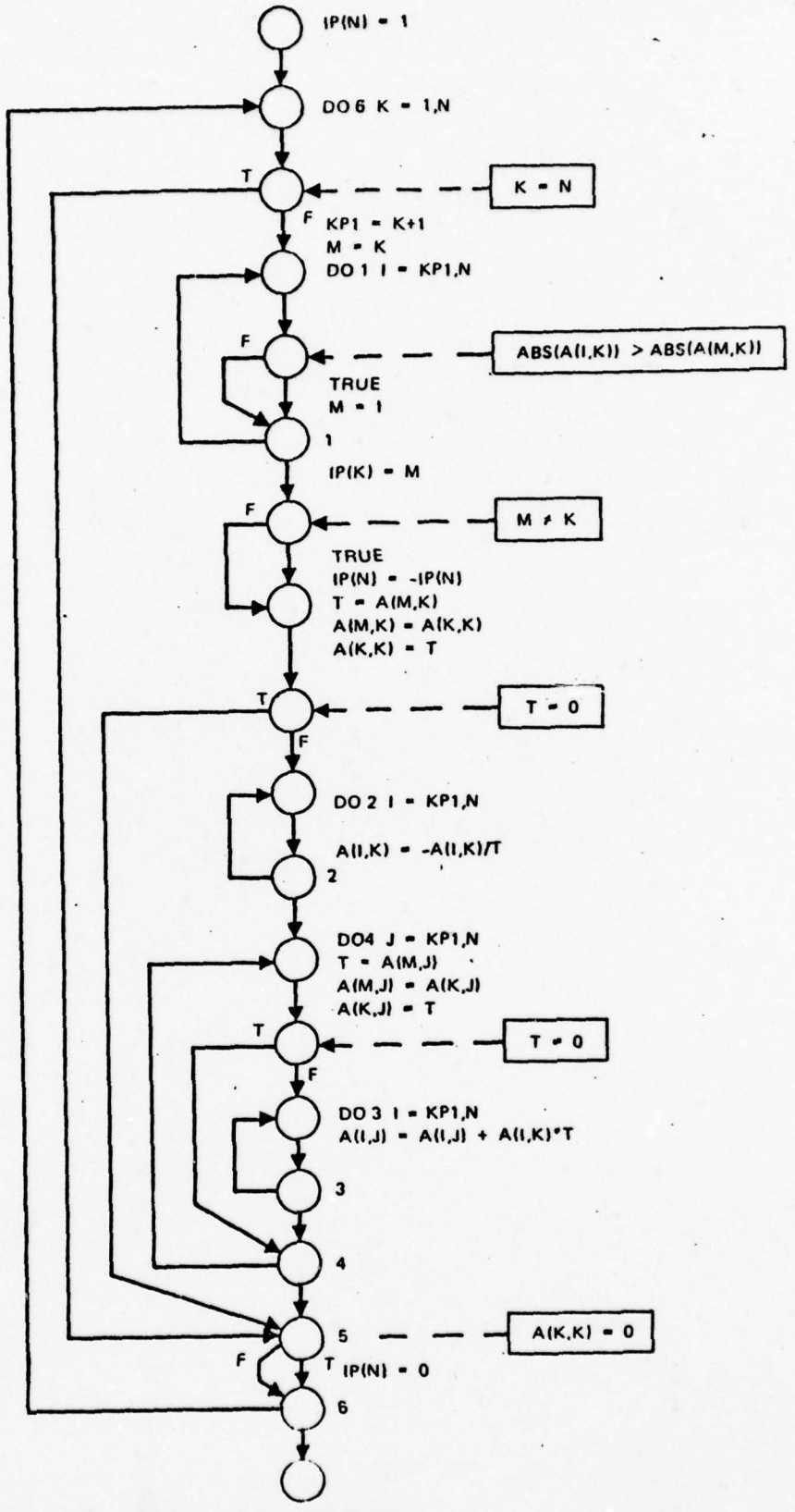


Figure 2. Combined Flow Chart and Code of Example Program

c_1 = ABS(A(2,1)) – ABS(A(1,1)) > 0

c_2 = ABS(A(3,1)) – ABS(A(2,1)) > 0
T = A(3,1)
A(3,1) = A(1,1)
A(1,1) = T

c_3 = ABS(T) > 0
A(2,1) = -A(2,1)/T
A(3,1) = -A(3,1)/T
T = A(3,2)
A(3,2) = A(1,2)
A(1,2) = T

c_4 = ABS(T) > 0
A(2,2) = A(2,2) + A(2,1)*T
A(3,2) = A(3,2) + A(3,1)*T
T = A(3,3)
A(3,3) = A(1,3)
A(1,3) = T

c_5 = ABS(T) > 0
A(2,3) = A(2,3) + A(2,1)*T
A(3,3) = A(3,3) + A(3,1)*T

c_6 = ABS(A(1,1)) > 0

c_7 = ABS(A(3,2)) – ABS(A(2,2)) > 0
T = A(3,2)
A(3,2) = A(2,2)
A(2,2) = T

c_8 = ABS(T) > 0
A(3,2) = -A(3,2)/T
T = A(3,3)
A(3,3) = A(2,3)
A(2,3) = T

c_9 = ABS(T) > 0
A(3,3) = A(3,3) + A(3,3)*T

c_{10} = ABS(A(2,2)) > 0

c_{11} = ABS(A(3,3)) > 0

Figure 3. Augmented Code for Example Program

It is, of course, possible to express the C's to explicitly relate them to the input data. This was done by Miller and Spooner threading back from the predicate, where variable is defined, through intermediate assignments to the original input data. This is fairly simple because the program is straight-lined. The technique is illustrated by a detailed analysis of the auxiliary variable, C_7. In terms of program variables.

$$C_7 = \text{ABS}\ (A(3,2))-\text{ABS}(A(2,2)),$$

and these can be traced through the calculations and assignments as follows:

substituting for A(3,2) and A(2,2),

$$C_7 = \text{ABS}\ (A(3,2)+A(3,1)*T)-\text{ABS}(A(2,2)^0+A(2,1)*T);$$

then, since only $A(2,2)^0$ is input data (and is marked by a superscript, 0) further backing is required; since $T = A(3,2)^0$ at this point in the program, and $A(3,2)^0$ is input data, the expression can be written

$$C_7 = \text{ABS}\ (A(3,2)+A(3,1)*A(3,2)^0-\text{ABS}(A(2,2)^0+A(2,1)*A(3,2)^0);$$

but $A(3,2)=A(1,2)^0$, $A(3,1)=-A(3,1)^0/A(3,1)^0$ and A(3,1) in the numerator is equal to $A(1,1)^0$. These and similar substitutions provide

$$C_7 = \text{ABS}\{A(1,2)^0- (A(1,1)^0/A(\#,1)^0)*A(3,2)^0]\}-\text{ABS}\{A(2,2)^0$$

$$-[(A(2,1)^0/A(3,1)^0)*A(3,2)^0]\}.$$

This is an explicit representation of C_7 in terms of input.

Although this process is feasible for simple programs, and in many respects resembles symbolic execution in reverse, it presents the same difficulties accompanying the development of equivalence classes. An alternative is to work forwardly from the input data to valuations of the C's, and their associated predicates. In this procedure, for properly picked input, all of the C's will be positive and the execution path will proceed along the prespecified path.

The new problem is then one of searching for areas rather than solving for points. These may seem to be problems of the same order of difficulty but they are not. In general applications the searching process need not proceed to the same level of definition that the solving process does. An analogy can be made with polynomial evaluation: it is far easier to locate a point where a polynomial is positive, than it is to find a root for the polynomial.

2.2.2 Test Techniques

To illustrate some of the characteristics of the test techniques employed the problem discussed above is taken in the framework of the flow diagram of Figure 4. The node numbers shown are in a 1-1 relation to the instructions and labels of Figure 2. DO-loops are easy to identify by the letters E (end) and S (stay), emanating from the end of the loop. The predicates are also easy to identify by means of the T and F letters labelling the exits. The DO1 loop, for example, starts at node 6 and ends at node 9, similarly the DO6 loop starts at 2 and ends at 31. The specified path for the sample problem can be identified in Figure 5. All predicate valuations (that are input dependent) are false except the one inside of the DO1 loop. Both True and False branches were shown to be taken of the nodes 3 and 11, corresponding to the predicates $\boxed{K = N}$ and $\boxed{M = K}$. These are not assigned auxiliary variables but are used to straightline the program; as a result they are permitted either predicate valuation.

Miller and Spooner employ several "objective" functions, generically denoted $f(C_1, C_2, \ldots, C_m)$; each has the property that $f>0$, when one or more of the C's is negative, and $f>0$ when all of the C's are positive. As an example, the function

$$F(C_1, C_2, \ldots, C_m) = \min(C_1, C_2, \ldots, C_m)$$

would serve for that purpose.

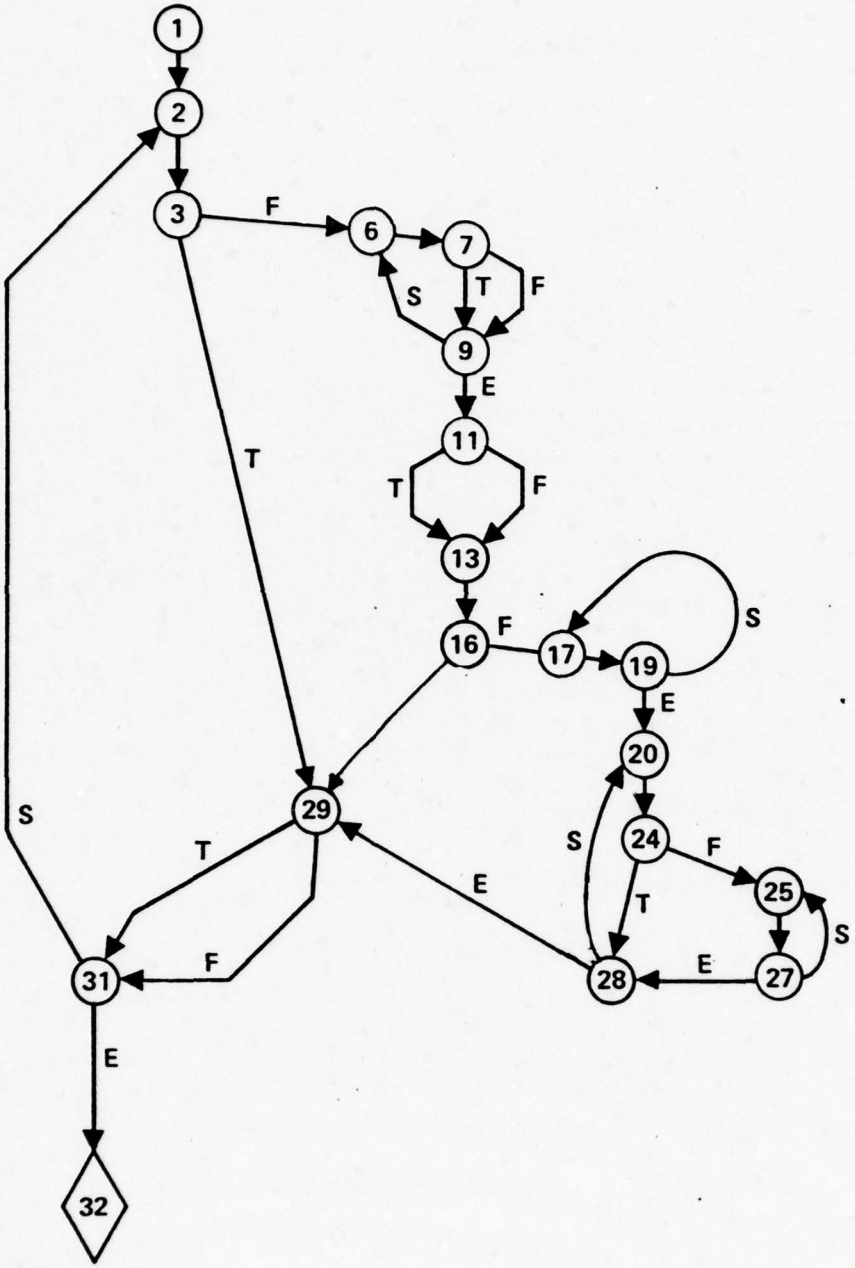


Figure 4. Flow Diagram of Miller and Spooner Example

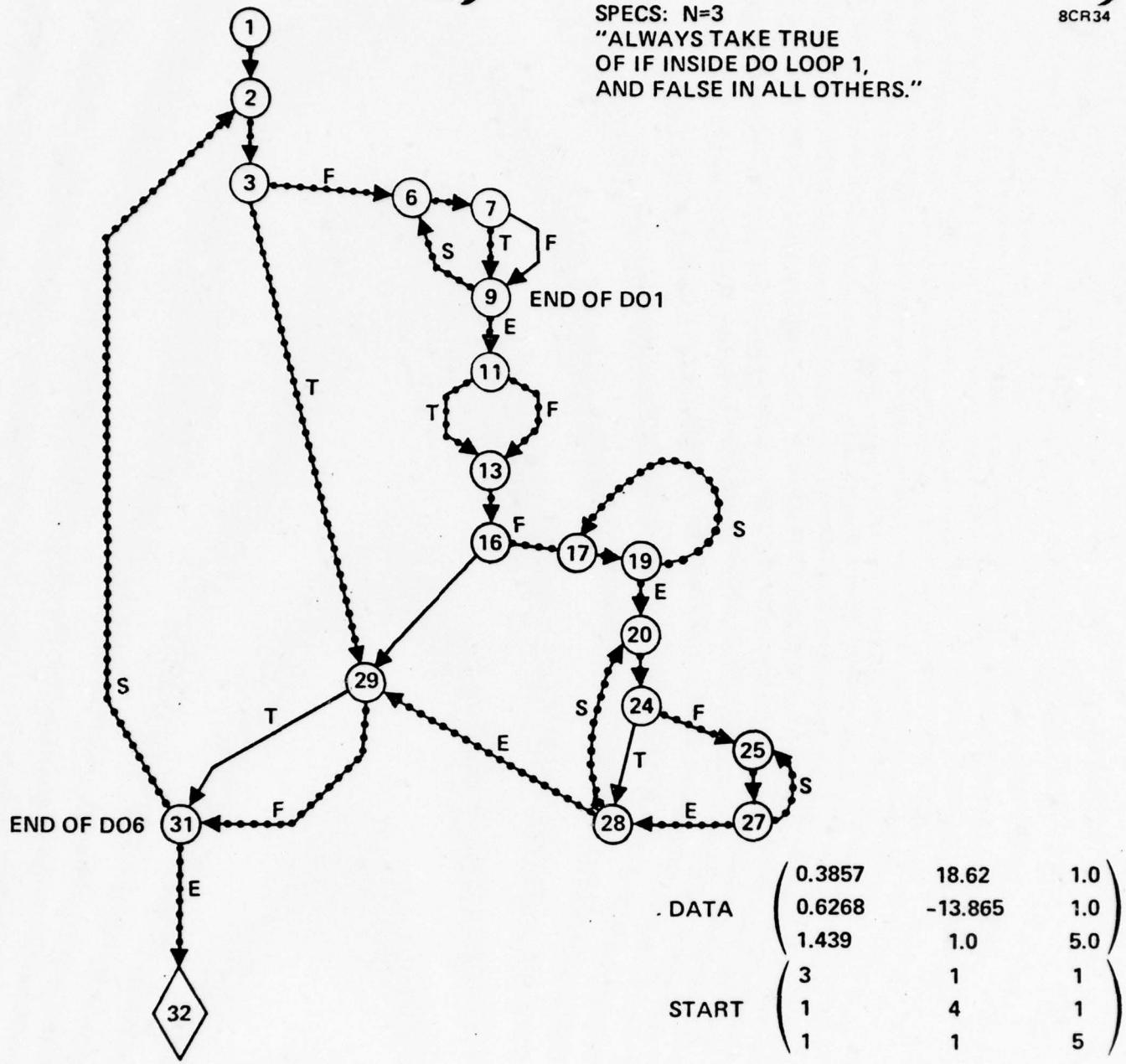
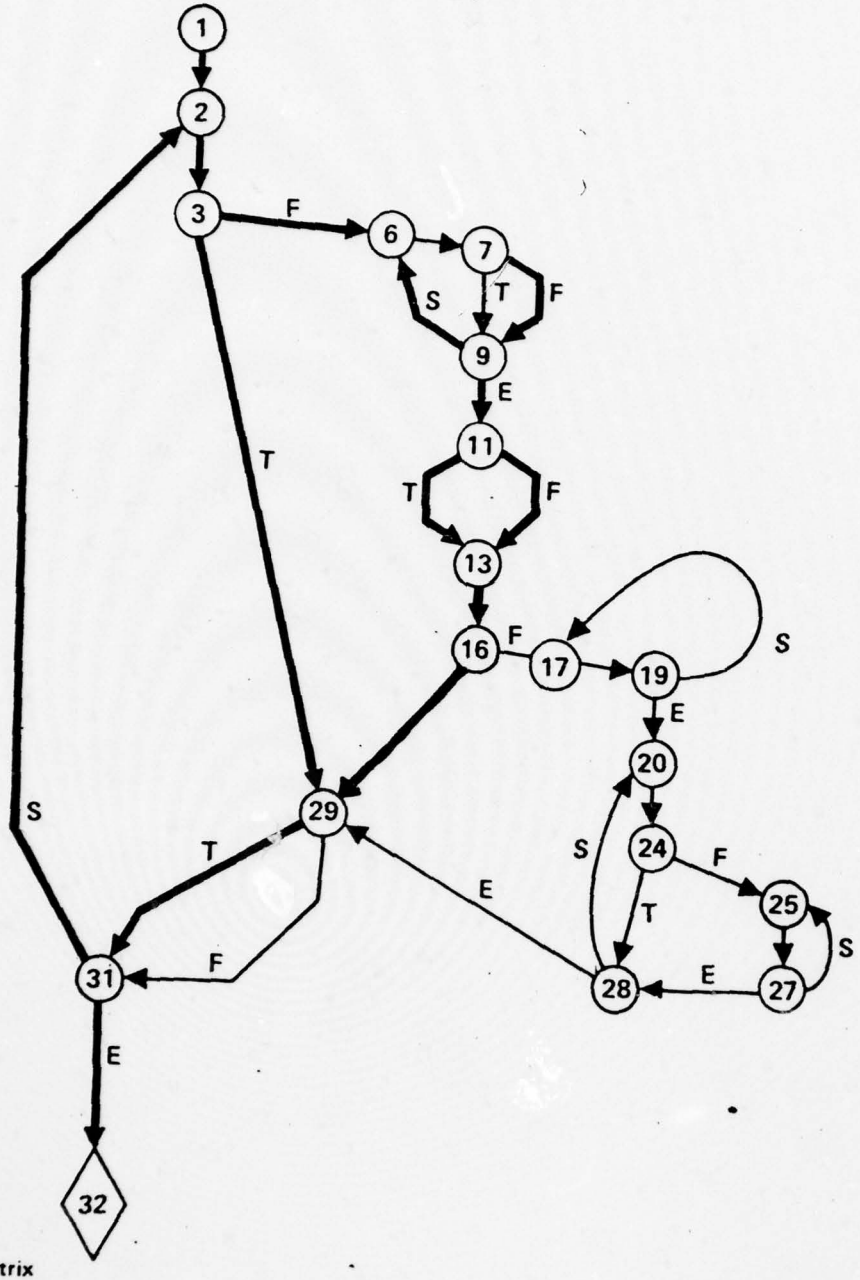


Figure 5. Selected Path Through Example Program



DATA $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$

Figure 6. Response to Zero Matrix

The problem at hand, then, becomes one of searching over the input data space for values where f is positive for the specified execution track. In the example problem, Miller and Spooner start the search with a "randomly" chosen matrix

$$A_0 = \begin{pmatrix} 3 & 1 & 1 \\ 1 & 4 & 1 \\ 1 & 1 & 5 \end{pmatrix}$$

which produces f = -2

Using direct search methods, they derive a data set

$$A = \begin{pmatrix} 0.3857 & 18.62 & 1.0 \\ 0.6268 & -13.865 & 1.0 \\ 1.439 & 1.0 & 5.0 \end{pmatrix}$$

which makes f=0.2411. According to Miller and Spooner, this is accomplished in less than 1 second of CPU time on an IBM 370/168. The resulting coverage of the program is indicated in Figure 5. Because of multiple passes over some portions of the program, depiction is less than perfect. The specified path, however, is achieved by the data.

The usefulness of this procedure is best appreciated when used in conjunction with a combination of the "random" drivers, suggested in the earlier work, augmented by constructed cases. The latter cases, are designed to "fill-in" for data that were taken so infrequently by random numbers that they make the former process uneconomical.

For illustrative purposes, the initial input data is taken as the "random" matrix, used by Miller and Spooner to start the process. For this matrix as input, the FALSE branch out of 7 is taken at least once. Thus, the "random" start exercises a path segment which the "optimum" data does not.

The predicates $\boxed{T = 0}$ are not true unless the zero-valued matrix elements. If the 3x3 zero matrix is used for data, all tests T=0, as well as the final A(K,K)=0 are true, and the constructed case produces the execution track shown in Figure 6.

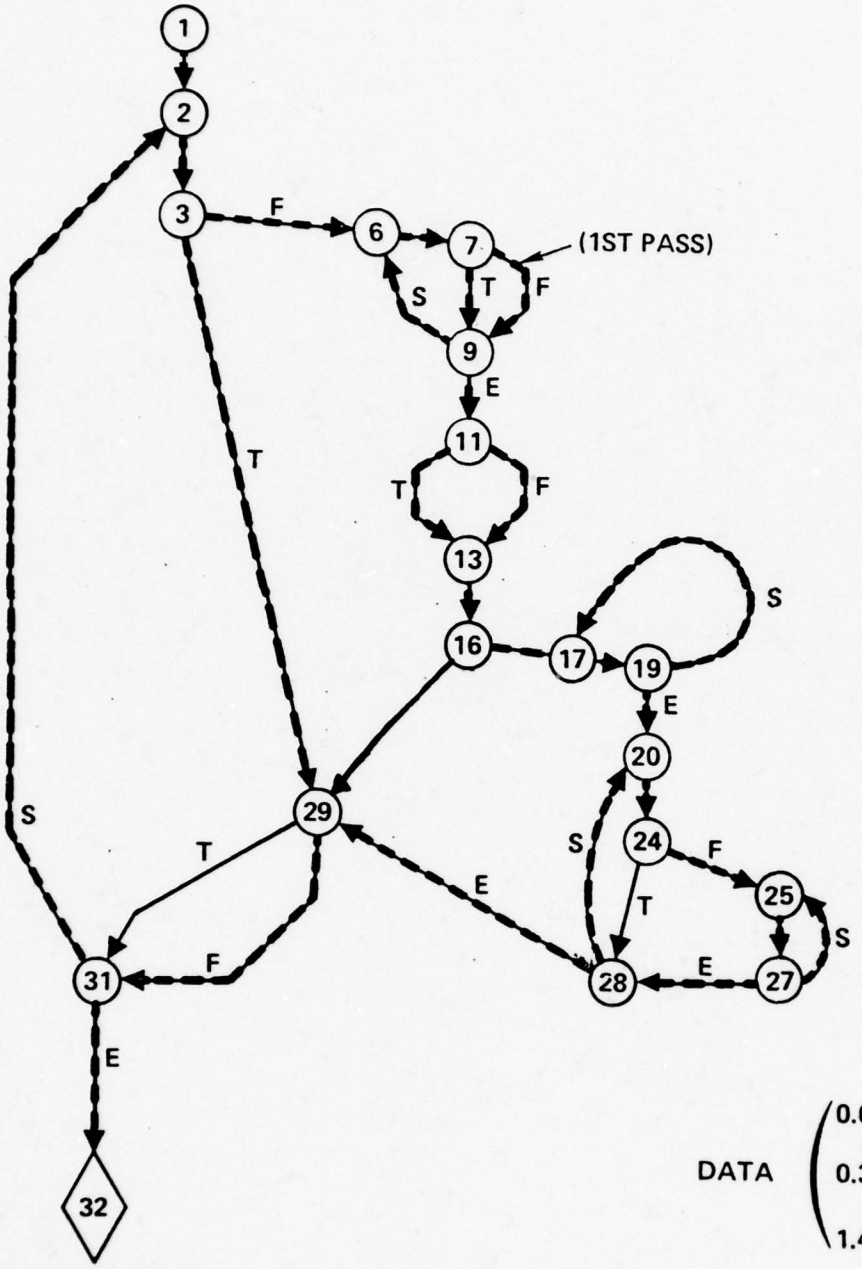
Additional tests for programs can often be suggested by some built-in symmetrics in the problem. Thus, for a short problem it can be generally assured that when input data is permuted, the resulting execution tracks will be different. When a polynomial solver is employed it is well known that a set of symmetric relations, involving the roots, define the coefficients of the polynomial. Further, there are relations between the coefficients of a polynomial and the polynomial whose roots are shifted, squared, and inverted. In the present instance of a matrix triangularization, the interchange of two rows can be expected to cause a different response.

As a matter of interest, when the matrix obtained by the optimization process is used with its 1st and 2nd rows interchanged, the resulting track is shown in Figure 7. The False branch out of node 7 is taken on the first entry, and the True branch on (one or more) subsequent passes. (In the particular sequence of tests employed, there is nothing new added by this test).

For the simple problem illustrated here, all segments of the program are tested by the three cases consisting of the starting "random" matrix, the zero matrix, and the matrix obtained by the optimization procedure.

The method suggested in the illustration leads to extensions of value to the general problem of exhaustive testing. As noted in earlier work, the problem of testing a program, only to the point where every instruction and every branch has been executed, is generally a computationally small enough problem making it feasible for almost any program. This is true basically because, for a program with k predicates (two-way), there are no more than 2k data points required to "test" the program in this way, whereas there are as many as 2^k different logical paths (many more if loops are permitted).

For the general testing problem, a sequence of random numbers or vectors may be used to develop a set of tracing vectors whose components represent the Boolean valuations of the C's. These runs would generally be both inexpensive and, because they are the first to be employed, would be of high yield. After a reasonably large set of random numbers have been run, the set of associated vectors (as distinct from the values of the auxiliary function exemplified by f in the above discussion) can be examined.



$$\text{DATA} \begin{pmatrix} 0.6268 & -13.865 & 1.0 \\ 0.3857 & 18.62 & 1.0 \\ 1.439 & 1.0 & 5.0 \end{pmatrix}$$

Figure 7. Response to Data Formed by Interchanging 1st and 2nd Rows of Original Matrices

Except for cases where predicates involve equality between expressions involving program variables, the vectors can be collected on the basis of component comparisons. Thus, if there are both zeros and ones* in the first component position, the testing has "exhausted" the cases provided by the first predicate.

A simple sorting procedure will identify unexercised branches. In case specific predicates are not represented by both "true" and "false" values, the process described above can be used to search for data that will force the program in the desired way. Should there be neither valuation, the same general procedure can be used initially.

It is possible in this scenario, that the so-called "scaling problem", a result of non-common scales on the variables involved and tends to confuse some optimization problems, can be used to advantage in the case of a search for data to exercise a specific branch. For example, if a variable associated with a predicate is made sensitive by multiplication or division of appropriate factors employed in its definition, then strong responses will occur with only small changes in input. A sequence of applications of such factors to each localized variable would, probably, produce good coverage.

The major use of the above technique is in establishing exhaustive tests for a given program package. The utility as a software metric is clear. As noted in Reference 10, one quality of software having universal appeal, is the degree to which a problem has been tested. Ideally this would be measured in terms of the ratio of the number of logical paths executed by all tests performed on the package, to the total number of paths present. However, the latter is almost never known, and there are many non-realizable paths which are not apparent; even the realizable ones may not be easy to enumerate. Thus the more easy to obtain ratio is a substitute.

---

*A blank would indicate no test.

Reference 10 describes the method of estimating the total number of paths realizable by random numbers. This method depended on the development of the count of the number of trials between discovery of new paths. An asymptotic limit to the total was then developed on the basis of an algorithm. This technique could be applied to individual branches or to any selected set of branches. Some measure of the degree to which a program has been tested may be developed from the combination of the yields obtained by using constructed cases and from application of random numbers. In specific production-type applications, studies of so-called impossible pairs may be made, but for development of a universal metric, such a fine-grained investigation is not warranted.

The primary difficulty in employing the Miller and Spooner technique is the construction of a straight line version of a program because it requires a specification of values for all predicates on each encounter, and tracing through all loops of the program. This may not be practical for even moderately large programs. Some variation of the process is needed and the Program Testing System (PTS), described in Reference 10, seems promising. This system instruments a FORTRAN program so that the code sections or segments, that a given data set exercise, can be identified. The branches from all decision points (or predicate tests) are identified by the PTS.

As noted in Reference 10, the identification of the particular execution sequence a program takes in response to an input set, is difficult to make primarily because certain segments are generally used many times and the sequential detail required for establishing an execution sequence, becomes lost in the summary statistics. It is always possible to identify the segments making up a logical path, and to count the number of times each segment is executed (in cases of loops), but there is generally no unique sequencing that can be established from the data; sequences AABACCD, ABAACDC, AAABDCC all have the same usage counts.

One of the practical aids in establishing an execution sequence and in guiding the testing, is to insert* variables of the type represented by C's in Miller and Spooner technique, and provide valuations as part of the statistics prepared by the PTS post-processor. The procedure of "straightlining" is then avoided, and the particular values taken at the program's predicate test point can be used to guide the choice of additional data.

---

*In case of loops it will be necessary to "create" these at the beginning of each pass.

2.3 EVALUATION OF SOFTWARE METRICS

2.3.1 Software Science Metrics

In the review paper by A. Fitzsimmons and T. Love (Reference 15) the principal metrics employed in Software Science are discussed in some detail. They are few in number: length, volume, program level, language level, effort and time.

A troubling feature of these metrics is that they are all based on counts of operators and operands and, as noted in the review, there are many cases where it is not at all clear what particular mix of these fundamental elements a given program instruction represents. The effects of this lack of precision in the definition of operator and operand has been studied by J. L. Elshoff (Reference 29). In this study all of the primary metrics are computed for some 34 different programs for each of eight different interpretations of the way in which the counts of programming elements should be taken. These eight different methods produced exceptional variability in the metrics in cases where there was a significant effect in the vocabulary definitions. For example, program No. 13 which is the largest program, showed counts of 185 and 746 for operations and operands, respectively, under the first interpretation, and counts of 118 and 900 for the second interpretation. The effects on the metrics under the two interpretations are:

estimated length	9,645	versus	8,512 (11.7% smaller)
volume	91,902		82,373 (10.3% smaller)
level	0.00365		0.00212 (41.7% smaller)
minimum volume	334.6		174.2 (47.9% smaller)
effort	25.243		38.945 (54.2% larger)
global level	1.1037		0.607 (45% smaller)

The variation in these metrics is indicative of the effect that the subjective choices (8 different types) can cause. In a separate comparison, the single metric, effort, for the 8 options (for program No. 1) were: 0.783, 0.881, 0.937, 1.010, 1.065, 0.764, 0.794, 0.679. This variability, which is over 50% (from min to max) is evidence of a lack of "objectivity" in this (and other) measures.

2.3.1.1 Complexity (Software Science Interpretation)

In the abstract of the paper by Fitzsimmons and Love it is noted that complexity of programs can be measured by the theory of Software Science. It was difficult to locate precisely where in the paper this complexity is measured because the word appears only incidentally in the text. It was determined, by direct inquiry, that it was measured by the effort metric.

This use of an extensive measure for complexity is indeed novel and does not correspond to intuition or to any other measures advanced by others. Halstead states that complexity of a program is measured by the total number of elementary discriminations required to produce it, and this count depends on the bulk of the program more than on its logical structure.

The previously published measures of complexity had to do with intensive measures such as the (normalized-to-unity) spectrum of the program listing across its indenture levels, or the density of branching statements.

The recently described measure of complexity by T. J. McCabe (Reference 17) is the cyclomatic number obtained from the flow graph of the program. This metric is described in a later section when the topic of complexity is re-examined. Suffice it to say, it is more an intensive measure than an extensive measure, and as McCabe points out (op.cit.) it is easy to write a program that is physically small but ultra complex.

The complexity measure of software science is directly related to the length of the program (the total number of operators and operands) and is finally developed on an absolute basis by the use of the so-called Stroud number, which is taken by M. Halstead to be 18 mental discriminations per second.

This Stroud number has, as its basis, some physical measurements of a human's ability to discriminate the frames of a kaleidoscopically presented visual sequence of images (related to the "flicker rate" in motion pictures). The use of this visual discrimination rate, as equal in value to the mental discriminations rate, is surely questionnable.

2.3.2 Software Metrics

Probably the best starting point for this discussion is a review of the metrics presented in the book by Tom Gilb (Reference 18). This serves more to cover the field than to make precise the concepts and definitions of the many metrics identified. Following this is a list of metrics having a reasonable likelihood of surviving through test and time.

2.3.2.1 Review of Gilb Metrics

2.3.2.1.1 Maintainability

The first definition offered by Gilb is that of maintainability. He defines it as

> "the probability that, when maintenance action is initiated under stated conditions, a failed system will be restored condition within a specified time."

That definition is essentially the same as that used for hardware. In the hardware case the measure is almost always applied in a bottoms- up way, that is the maintainability is derived for each major assemblage from the records of its contained minor assemblies; the system's figure is derived from the major assemblies. Work records on the times to fix are estimated during design, and, once hardware is delivered, records are kept of the actual fix times.

Software should be amenable to the same broad guidelines. Some modules are likely to be more easily fixed than others and a better systems-wise figure can be developed from the bottoms-up composition. The records of maintenance of individual modules should be used to extrapolate for new errors. The fact that the process of error-findings tends to have long periods between finds (probably) does not alter the fundamental measure of the average time to fix. This is (probably) so because the late occurring errors are (probably) not of a different level of difficulty than the early occurring errors. (Should there be a trend towards longer fix times with the "age" of the error, a model would need to be developed).

2.3.2.1.2 Logical Complexity

Gilb's introduction into this topic identifies early work by L. Farr and H.J. Zagorski, who used the IF statement density as a measure of the logical complexity. Gilb also mentions "psychological" (his quotes) complexity of source programs and refers to some statistical work by L. M. Weissman which correlated metrizable program aids (comments, indentations, etc.) to productivity and accuracy.

2.3.2.1.3 Structuredness

One of the metrics identified by Gilb is structuredness. This was one of many metrics proposed by TRW in a study for the National Bureau of Standards.

Structuredness is one of 12 low-level metrics identified by Gilb, the others are: device independence, completeness, accuracy, consistency, device efficiency, accessibility, communicativeness, self-descriptiveness, conciseness, legibility, and augmentability.

For structuredness, there are 9 submetrics which are in actuality, questions concerning the existence of module size limits, program flow, and so forth. Gilb's Figure 51 (page 103) can be referred to for identification of the particular questions. It does not appear that the underlying metrics have any quantitative basis, and necessarily have either a zero or an all value.

Typical of a question, under a column headed "Definition of Metrics to Measure Structuredness," is: "Do all subprograms and functions have only one entry point?" Here, should the answer be no, there is no way of differentiating between "all-but-one" and "none."

Presumably a yes answer to all questions would indicate a perfectly structured program. Using these the characterizing features (from Figure 51 of Gilb) the program would be one which:

A. Has rules for transfer of control between modules.
B. Has limited modules sizes (Note: the limit is not specified).

C. Has the ordering: commentary header block, specification statements, executable code (Note: it is hard to imagine a program that does not follow the order)
D. Subprograms all contain at most, one point of exit
E. Subprograms and functions all have only one entry point
F. Program flow is always forward, except where commented
G. Overlay structure is consistent with the subprogram's sequencing
H. Is subdivided into modules in accordance with readily recognized functions.
I. Is written in standard constructs

These submetrics are then scored as to their "correlation" with a "high score for the metric." The use of "correlation" as a descriptor for subjective judgement is highly questionable: there are no numbers to associate with the identified metrics, and the numbers associated with the "score," if present at all, are certainly vague.

Nonetheless, the "quantifiability" of the metrics is judged against six categories which, while neither exhaustive nor mutually exclusive, are nonetheless indicated as such by the tabular entries.

The other 12 metrics are probably treated in the same way as the structuredness metric, and, beyond their identification, do not appear to merit additional inquiry. (Self-descriptiveness, communicativeness, and accessibility, for example, appear to be invented to exercise the invention process, and do not represent useful metrics; others, such as augmentability, may have some value).

2.3.2.1.4 Reliability

Gilb's definition of system reliability is in close accord with the customary (hardware) definition. It states that "reliability is the probability that the system will perform satisfactorily (with no malfunctions) for at least a given time interval, when used under started conditions." This is modified only slightly under the definition offered later. Gilb's variation of his definition for system reliability when applied to program or software reliability are minor, a particular machine is denoted, and operations are "within design limits."

2.3.2.1.5 Repairability

The concept of repairability is a variation of the maintainability concept, the emphasis is on the probability of a repair within a specified time, when maintenance is performed under specified conditions. The requisite tools parts and men, are assumed to be available at the start, and this is one of the specified conditions.

2.3.2.1.6 Serviceability

This metric is taken from hardware reliability and is the degree of ease or difficulty with which a system can be repaired. It is not considered quantifiable at present.

2.3.2.1.7 Availability

Again, from the hardware reliability definition, this is the probability a system is operating satisfactorily at any point in time. It is usually measured by a ratio of times or mean times, and Gilb offers three variations of the concept (intrinsic-, operational-, and use-availability).

2.3.2.1.8 Attack Probability

This metric is one of several Gilb suggests in the security aspects of programs. This metric is the probability of an attack (of a particular type) on a system during a particular time interval.

These attacks can be considered to be active (malicious) or passive (typified by invalid data).

2.3.2.1.9 Security Probability

This is described by its alternative title, attack repulsion probability, and is a metric gauging the probability of a successful rejection in the system at any time. The attack type is specified. Gilb states that this concept is close to the concept of error detection probability. This is less true of active attacks (which may not persist) than it is for passive attacks such as bad data.

2.3.2.1.10 Integrity Probability

This probability is the probability of no successful attack on the system:

$$I_g = 1-[A_t.(1-S)]$$

where A_t is the attack probability for a particular time interval, and S is the probability of rejection (for all times).

2.3.2.1.11 Accuracy

Several examples of the metric and a discussion contrasting it with precision are given by Gilb. The measurement ratio, correct data/all data, appears to be too vague for use involving, as it does, the idea of "correct data". Usually accuracy involves a continuum of values so that "correct" data is too narrowly defined for practical usage.

2.3.2.1.12 Precision

The suggested measure of this metric, which aims to gauge the degree "to which errors tend to have the same root cause," is the ratio formed by dividing the number of actual errors at source, by the number of corresponding root errors observed in total caused by source bugs.

The difficulties in first knowing how many errors there are at the source seem unsurmountable, and tying together the "corresponding" errors with the source would not seem to be an easy task.

2.3.2.1.13 Error Detection Probability

Gilb suggests a categorization of the error types and an assignment of the likelihood of detection of errors of the pre-specified type. The failure to include time aspects into the problem makes for a flawed definition. The probability of an eventual detection of an error is (probably) unity for almost all error types.

2.3.2.1.14 Error Correction Probability

As defined by Gilb, this is the probability of reconstructing "data in the form and content originally intended." This is a vague concept when identification of the random event is sought. The originally intended form and content is generally not known, rather it develops as effects are judged unsatisfactory and tentative changes are made. There is a chance that the repair made will have an error that may lie undiscovered for a period of time, and so time should be involved in the measure in some way.

2.3.2.1.15 Logical Complexity

In the text two metrics for logical complexity are identified, the number of binary decisions and the ratio of absolute logical complexity to total complexity. But Gilb also suggests under the Figure 83 on page 161 that it be measured by the number of possible logical path combinations in a program.

In this respect Gilb illustrates with an unanswered question, the defect in using even the density of branching statements as a measure of complexity. In his Figure 84, two programs are shown, one which has 6 binary decision points and the other only one. But for a sufficiently large number of total instructions (say 239 as indicated in the description) the density of the clearly more complex program is less than the ultra-simple one. This alert is examined in the later discussion of complexity.

2.3.2.1.16 Flexibility

Gilb defines this as that part of complexity that is useful, and it is the ratio of useful to total that is the metric.

2.3.2.1.17 Built-in Flexibility

This is deinfed as the ability of a program to immediately handle different logical situations. It must be built-in in order to respond without loss of time.

2.3.2.1.18 Adaptability (open-ended flexibility)

Gilb acknowledges the difficulty of originating a metric for this concept and suggests, as a tentative measure, the count of the linkages between modules. This is the same as the metric used later for structural complexity.

2.3.2.1.19 Tolerance

This is defined as the ability of the system to accept different forms of the same information as valid. The proposed metric is the count of the number of different variations that can be handled by the system, where variation means the different media, different formats of input, or logical variations (such as misspellings and synonyms).

2.3.2.1.20 Generality

The "degree of applicability of a system within a stated environment" constitutes generality. Its measurement is subjectively assigned (0 to 1).

2.3.2.1.21 Portability

This is defined as the ease of conversion of a system from one environment to another. The metric is obtained by first forming the ratio of the resources required to move the program to a target environment to the resources needed to create the program for the target environment, and then subtracting the ratio from unity. The result is the ratio of the cost difference to the creation cost and, on the extremes, agrees with an economic measure of portability, because for a zero-cost move the portability is unity, and for a cost equal to the creation cost, the portability value is zero.

2.3.2.1.22 Compatibility

This attribute is, according to Gilb, related to the concept of portability, the difference being that portability is a characteristic of a single system whereas compatibility applies to an average over a class of systems. This distinction provides the metric, an average portability over the collection of program systems.

2.3.2.1.23 Redundancy Ratio

This is the first of what are called structural metrics by Gilb. This ratio generally is formed by taking the actual count of quantities to the minimum possible count.

2.3.2.1.24 Hierarchy

This structural metric describes the number of indenture levels and the spectrum of program elements across these levels.

2.3.2.1.25 Structural Complexity

As noted earlier in the section concerning adaptability, this is measured by the number of modules(absolute) or the ratio of linkages to the total number of modules. This is an easy metric to derive for some languages as Gilb shows. For FORTRAN the modules are counted by the number of subroutines and functions, and the number of linkages is the total of subroutine parameters and the references to the common area.

2.3.2.1.26 Modularity

Although modularity is stated to be a synonym for structural complexity, it seems to stress the number of modules and not the linkages.

2.3.2.1.27 Distinctness

Distinctness as defined by Gilb involves errors and, in fact, is measured by a ratio between the number of bugs in the module and the number that are common to the module and another ("simultaneously"). It is hard to see how this ties to the intuitive concept of uniqueness, particularly how errors are necessary components of distinctness.

2.3.2.1.28 Effectiveness

Among the performance metrics, effectiveness is listed first. It is a probability of "success" within a given time and specified environment. The "success" means meeting an operational demand. Gilb composed efficiency from three probabilities: reliability, readiness probability, and design adequacy (on a scale from 0 to 1).

2.3.2.1.29 Efficiency

This attribute is defined as the ratio of useful work to the total expended.

2.3.2.1.30 Cost

Among financial metrics are costs and its major subdivisions, fixed and variable. Gilb uses the terms capital and operational.

2.3.2.1.31 Time

Computer and "Human" time resource metrics.

2.3.2.1.32 Space metrics

This is more commonly called the size of a program. It can be measured on an atomic level by bits and bytes and, on the more common scale, by the number of instructions.

2.3.2.1.33 Information

Gilb says that information content of a program is not directly measurable, and suggests use of "useful data" as an indirect means for measurement.

2.3.2.1.34 Evolution

This is a measure of the incremental change to a system during a time interval, t. If the change is so pervasive that it constitutes a substitution, the metric would have a value of unit.

2.3.2.1.35 Stability

Stability is the complement of Evolution and it denotes the percentage of unchanged content of a program (over a specified time period).

2.3.3 Candidate Metrics

Clearly some of the questions that should have been asked of the community several years ago are:

A. Are any attributes worth study?
B. Which attributes are useful?
C. Can these be measured in a form useful to the community?

It is clear from inspection of the Gilb metrics that there are many that will not survive the tests required of practical gauges. Most of the 13 low-level metrics identified by Gilb have little hope of common usage. The discussion concerning structuredness, in that section, indicates that the concept is initially vague and becomes amorphous after its component parts are identified (in the form of questions).

Of the metrics listed above, the following are considered of primary value: reliability, complexity, cost and time.

Regarding as secondary in importance are: maintainability and availability.

Supplementing these metrics are some that history may judge to be of more value than any of the metrics identified above: mean-time-to-next error, mean-time-to-perfection, error content, testedness, and purification level.

Excepting the complexity metric, it does not seem necessary to amplify on the previous Gilb definitions, and the following subsections deal with the augmenting metrics.

2.3.3.1 Mean-Time-To-Next-Error

Of primary importance in the testing of programs is the decision on whether or not to release a given module or program. A good guide to this choice lies in the time pattern of the errors found, whether this pattern lies in a data base metered by CPU units, hours, days, or weeks is not relevant (except as its potential future uses may have to be considered). If the time pattern indicates a steady state or constant error rate, or, even worse, shows an increasing failure rate, there is clearly no reason for releasing the module and much evidence to the contrary. Once a pattern of decreasing counts (per unit time) is achieved, any of several models can be applied to the data that the error pattern represents, and estimate of the mean-time-to-next-error can be obtained.

It is the magnitude of this mean-time-to-next-error, or more commonly called the mean-time-to-failure, MTTF (which for a certain probability distribution, and steady state conditions, is the same as the mean-time-between-failures [MTBF]), considered in the context of its expected use, that is important. For real-time systems, governing, for example, weapons or aircraft, the MTTF should be several times as large as the mission duration. The proper figure for the MTTF is determined by the reliability specified by the customer for the system.

Values for the MTTF are available in any of several models: Jelinski-Moranda, Shooman, Schick-Wolverton, Moranda Geometric Purification, Moranda Hybrid Geometric-Poisson.

Littlewood and Verrall avoid MTTF and insist instead on percentiles (such as the median) of the distribution describing the time between errors.

It is important to note, that for all models the MTTF is a parameter that is changed by either event or time. The Jelinski-Moranda model has an MTTF, indexed by the dummy variable i, which increases at the occurrence of each error, and can be given in terms of the model parameters N and ϕ as

$$MTTF_{J-M} = \frac{1}{[N-(i-1)]\phi}$$

The Schick-Wolverton model has an "instantaneous" MTTF depending on both time and event, and it has an averaged MTTF that is obtained from the first moment of the Rayleigh distribution for the time of next error. Thus,

$$MTTF_{S-W} = \sqrt{\frac{\pi}{2}}\left[\frac{1}{(N-n)\phi}\right]^{1/2}$$

For the Geometric Purification model, the MTTF is

$$MTTF_{G-P} = \frac{1}{Dk^n}$$

where D is the failure rate for the first error, k is the geometric ratio which is used to obtain the error rates, and n is the number of found errors.

The Shooman MTTF is given by

$$MTTF_S = C\left[E_T - E_c(t)\right]^{-1}$$

where C is a proportionality constant, E_T is the total error content.

2.3.3.2 Mean-Time-To-Perfection

Some models permit an estimate of the mean time required to achieve an error-free program. Generally this estimate is accompanied by a variance (standard deviation) that is so large that it has little or marginal utility. It is nonetheless a guide to management and it is changed, and generally made more precise, as more errors are discovered.

The simplest way to form this estimate is to sum the estimated MTTF's for all remaining errors; hence (using MTTP for mean-time-to-perfection), the estimate so formed for the Jelinski-Moranda model is:

$$MTTP_{J\text{-}M} = \frac{1}{\phi} \sum_{j=n}^{N-1} \frac{1}{N-j}$$

For the Schick-Wolverton Model,

$$MTTP_{S\text{-}W} = \sum_{j=n}^{N-1} \sqrt{\frac{\pi}{2}} \left[\frac{1}{(N-j)\phi} \right]^{1/2}$$

This last formula is incorrectly given in the latest Schick-Wolverton paper (reference 19).

The Shooman model does not permit an estimate of the MTTP because the failure rate of that model is a continuous exponential. The mean time to achieve a zero with the exponentially decreasing failure rate is infinite.

The Moranda Geometric Purification model also does not have a finite average time perfection. Even though discrete, the failure rate does not attain a zero value.

The Littlewood and Verrall model, based on Bayesian adjustment, does not involve a parameter that can be directly related to the MTTF, and it is required that some alternative be found. This can be provided by any of the percentiles of the distribution formed by convolution of the family of related exponential distributions they use in their examples. It is necessary, however, to rely on the most recently available, a posteriori distribution for one of the two parameters, and to continue the assumption concerning the way that the sequence of values for the other parameters are related. It presents a difficult problem analytically and probably has practical objections.

The recent publication by A. L. Goel and K. Okumoto (reference 20) has relevance to this and some of the other problems. Their work in the present context employs a family of distributions that are the same as those used by Jelinski and Moranda but with an essential difference, they assume an imperfect repair and account for it with a parameter, p, that is the same for all errors. Using these variations, the distributions of the "first passage" times (zero errors) and of the times to achieve various levels of purification are derived.

2.3.3.3 Error Content

Three models can be used to derive estimates of the error count. The Jelinski-Moranda model accomplishes it through use of equations developed from the assumption that there is a direct proportion between error content and failure rate. The corresponding Shick and Wolverton assumption is that the failure rate is proportional to both the number of errors and the "debugging time." The Shooman model can be used at two or more separated time intervals to estimate the error content. From observations of the average MTTF for these intervals, parameters of the linear relation between failure rate and error content can be found by simultaneous equations (for two intervals) or by least squares (for three or more).

2.3.3.4 Purification Level

Although some models do not measure error content and may not achieve a perfect state, there is a measure that, in some cases, can be used to describe the state of perfection achieved at a given point in time. For error-content models, the ratio of the number found to the total number (estimated) provides the reasonable estimate. For the Moranda Geometric Purification process, the purification state can be estimated by taking the ratio of the initial failure rate to the achieved failure rate.

The purification level or percentage is clearly of more value than the error content since the absolute number is, by itself, generally a poor indicator of status because it is size-of-program related.

The several estimators of the purification percentage are (in terms of their defined parameters):

Jelinski-Moranda	$\frac{n}{N} \times 100$
Schick-Wolverton	$\frac{n}{N} \times 100$
Shooman	$\frac{E_c}{E_T} \times 100$
Geometric Purification	$(1-k^n)(100)$

2.3.3.5 Testedness, Degree to Which a Program Has Been Tested

A metric of a different kind is represented by the degree to which the program has been tested. There are several different types of "coverage" for a program, where "coverage" means that the program "elements" have been executed.

E. C. Miller (reference 21) presented a useful list of several different coverage types in a sequence reflecting the increasingly larger size of the covering unit. The lowest level of coverage is obtained when every statement is executed at least once, the next level is achieved when each segment associated with the explicit or implicit predicate outcome is executed. For complex programs involving nested loops, the test coverage may necessarily be limited to the exercising of the program so as to test, one time, all so-called boundaries and interiors of loops, it being assumed that all segments are exercised. (Boundaries are the entries and exits from a loop.) A higher order of coverage consists of multiple passes through loops, these are tests that iterate all loops up to a certain specified limit (even 1), and provides additional coverage. The ultimate test coverage (with several other types in between) exercises all logical paths through a program.

One additional type is afforded by the testing of the type described in the earlier subsection, where tracks were identified. Coverage by tracks is intermediate between segment coverage and logical path coverage.

The metric for any of the types is simply the ratio of the number of "elements" (defined as instructions, segments, branches, predicates, tracks or logical paths) to the total number of these elements.

A new concept was introduced by Moranda (reference 10) where the difficulty of enumerating the number of different elements is avoided. By using random numbers, it is possible (under some assumptions that are reasonable or acceptable to some and questionable or unacceptable to others) to estimate the total number of program elements that will eventually be achieved by random numbers. This technique was used in the original work (reference 10) to estimate the total number of "tracks," but could as easily be used to derive the asymptotic limit to the number of logical paths.

2.3.3.6 Complexity

As noted in the discussion on the effort metric employed for complexity measures by the Software Science advocates, the use of an extensive measure for complexity runs counter to intuition. The total number of "elementary discriminations" required to produce a program does not seem to properly reflect the structural aspects of complexity, for a "straight-line" program (no loops) of extreme length would have a high effort value, but might be judged rather simple.

Other measures were suggested in that same discussion. The density of branching statements was suggested, but as noted in subsection 2.3.2.1.15 the density, as measured by instruction count, may be misleading. In that section, a reasonably complex program containing six branches had so many (hypothesized) instructions that the density of branching statements was less than a short straight-line program (with one branch).

It is clearly necessary to alter the concept, and base the metric on the segment counts, rather than the instruction counts. This is a reasonable position to take because some segments may contain a very large number of instructions. As far as the intricacies or complexities of a program are concerned, all segments are the same and do not depend on the number they are comprised of.

Thus, a more satisfactory metric for complexity would be either the number of segments or the number of logical paths. Since the latter are difficult to count in many cases, the former can be used, even though the way they connect is not measured thereby.

Another measure of complexity which may be of use is the indenture level spectrum. This concept is rather simple in that it tallies into each indenture level, each instruction of the program. By dividing the number in each category by the total number of instructions, a normalized-to-unity spectrum can be produced. There are clearly some deficiencies in this approach since a program that "shifts" back and forth between two adjacent levels is not judged to be more complex than one that has the same number of instructions at each level and "shifts" but once. The metric would required a complementary measure to provide a total measure of complexity.

A far better metric for complexity has been developed by T. J. McCabe (reference 17). He suggests that the program be represented by a directed graph, G, in the usual way. The way the nodes (or vertices) and segments (edges) of G are connected is measured by a cyclomatic number, denoted by V(G), determined by the number of edges, vertices, and connected components (where the latter is a subgraph of G).

McCabe proves a theorem that permits an alternative way of finding the cyclomatic number: for strongly connected graphs, the cyclomatic number is the maximum number of linearly independent circuits. In order to apply this theorem, it is necessary to form a strongly connected graph by looping back from the exit node to the entrance node.

It is generally easy to identify the cyclomatic number of most reasonably well-structured programs of small to moderate size. Where the program is extensive, the algebra set up by McCabe can be used to calculate the number.

3.0 PUBLICATIONS

1. P. B. Moranda, "Limits to Program Testing with Random Number Inputs," Proceedings of COMSAC 1978, November 13-15, Chicago, Illinois

2. P. B. Moranda, "Event-Altered Rate Models for General Reliability Analysis," accept for publication in the special issue on Software Reliability of the IEEE Transactions on Reliability.

3. P. B. Moranda, "Asymptotic Limits to Program Testing," to appear in INFOTECH State-of-the-Art Report on Program Testing, ed. E. F. Miller, 1978.

4. P. B. Moranda, "Probability-Based Models for the Failures During Burn-In Phase," under review for publication in the Journal of the Operations Research Society of America.

4. List of Professional Personnel

ZYGMUNT JELINSKI - Branch Chief, Computer Sciences

University of London:	B.S., Economics and Statistics (1952)
Metropolitan College, England:	M.B.A., Business Administration (1953)
University of London:	M.S., Mathematical Statistics (1954)

Mr. Jelinski has been managing computer research and development programs for 23 years. As Branch Chief, Computer Sciences, he currently directs research in software reliability, modeling, validation and verification, language processing, emulation, and simulation. Software tools developed and/or maintained under his direction include the Program Evaluator and Tester (PET), the SUMC Meta-assembler, the Compiler Writing System, the OPAL Development Tool (OPALDET), CAMIL, and the Generalized Language Processor (METRAN). He was Study Manager, or a contract to develop methodology for effective test case selection (a research contract for the National Bureau of Standards) and he directed a study for NASA on methodology for reliable software. Another program under his direction was the Company-sponsored Software Validation Study, during which the nature and circumstances of software malfunctions were determined and software validation methodology was developed and applied to a tactical software system, resulting in accurate prediction of software malfunctions. He also directed the Software Reliability Study sponsored by the Air Force Office of Scientific Research. This study involved research into the development and evaluation of mathematical models representing the pattern of software malfunctions. At Rockwell International, Mr. Jelinski was Chief of Systems Programming Technology. In this capacity, he directed the design and retrieval systems, and engineering design aids. Earlier, he managed all programming for the RECOMP II and III computers. At Philco Corporation, he was manager of Programming in support of Philco 2000 computer marketing, and at RCA he was Manager of Applications for RCA 4130 communication computers.

Mr. Jelinski's publications include the following:

HOLDET - Higher Order Language Development and Evaluation Tool (coauthor), MDAC Paper WD 2769, presented to Computers in Aerospace Conference, Los Angeles, November 1977 (AIAA,NASA,IEEE,ACM).

An Approach to Solution of Problems with Support Software as Deliverables, MDAC Paper WD 2759, presented to Defense Systems Management Review, Ft. Belvoir, Virginia, March 1978.

Recent Software Development Techniques in the United States, MDAC Paper WD 2706, presented to Polish Academy of Sciences, Warsaw, September 1976.

Software Reliability Predictions, with Dr. P. B. Moranda, MDAC Paper WD 2482, presented to the Federation for Automatic Control, Boston, and published in its proceedings, August 1975.

Can Statistics be Applied to Software - Historical Perspective, MDAC Paper WD 2531,presented to the Computer Science and Statistics 8th Annual Symposium on the Interface, Los Angeles, February 1975.

Applications of a Probability-Based Model to a Code-Reading Experiment, with Dr. P. B. Moranda, MDAC Paper WD 2067, presented to the Symposium on Software Reliability Sponsored by IEEE, New York, and published in its proceedings, April-May 1973.

Generalized Events-Oriented Simulation System (GESS) - A Performance Evaluation Tool, with Dr. G. S. Chung, MDAC Paper WD 2033, presented to Computer Performance Evaluation Users Group sponsored by National Bureau of Standards, Washington, D.C., published in proceedings, October 1972.

Software Reliability Research, with Dr. P. B. Moranda, MDAC Paper WD 1808, presented to the Conference on Statistical Methods for Evaluation of Computer Systems Performance, Providence, R.I., and published in its proceedings, November 1971.

PAUL B. MORANDA - TECHNICAL ADVISOR

AB, Chemistry, 1942 , Fresno State College
MA, Mathematics, 1948, Ohio State University
PhD, Mathematics, 1953, Ohio State University

McDonnell Douglas Position: Information Systems Advisor, Senior

Dr. Moranda was the Principal Investigator of a contract with AFOSR to investigate the development of Quantitative Methods for Software Reliability. He is also working on Software Validation. During the first nine months of his employment at MDAC he was Principal Investigator for the Software Reliability IRAD. During this period he developed mathematical models for software discrepancies and applied it to failure data to obtain estimates of the error content of software packages, and estimates of their period of error-free performance. For one year subsequent to that assignment, he analyzed and developed logistics model for the YC-15 STOL aircraft.

Previous Experience: Prior to joining MDAC in 1971, Dr. Moranda was Manager of Systems Analysis at Computer Real Time Systems of Newport Beach. Prior to this, he was employed by North American Rockwell for a period of six years. Before joining Computer Real Time Systems, he was Technical Advisor to the Director of Data Management Systems at Autonetics' Information Systems Division. During this interval he participated in several systems studies and in-house development efforts. In the field of transportation, he was responsible for developing a system framework for the analysis of advanced marine transportation systems. Additionally, he participated in: a quantitative analysis of the operations of the over-the-counter trading department of Merrill, Lynch, Pierce, Fenner and Smith; an overview study of the American Stock Exchange; and a systems analysis of the U.S. Federal Court System.

In 1967 he was appointed Scientific Advisor to the Director of Management Systems in which capacity he developed methods of economic forecasting of sales and other business parameters employing random wavelet concepts. This work led to the formulation of a simulation model which predicted, in a balanced way, the sales, profit, cash flow, headcount, backlog, and facilities requirements for the entire corporation.

He devoted full time to work on the California Integrated Transportation Study, performing system synthesis, trade-off studies, and mathematical analyses. In recent years, he has presented five lectures in the transportation field at leading universities: in October of 1966, he presented a lecture on "Advanced Concepts in Transportation Planning" at Carnegie Institute of Technology; in June of 1966, and again in June of 1967, he lectured on the "Application of Systems Analysis to Large Scale Systems - Transportation" at the University of California at Los Angeles; in the Spring quarter of 1967 he organized and administered a full time upper division course in Dynamic Modeling for the University of California at Berkeley, in which he also delivered two lectures, including a summary of the California Transportation Study and the application of analytical techniques to the study of transportation problems.

Prior to joining Autonetics, Dr. Moranda was at the Aeronutronics Division of the Philco Ford Company, engaged in operations analysis of various projects. On a special assignment to the Ford Motor Company, he was responsible for mathematical modeling of the complete automobile production process. Other assignments included development of a methodology for handling fragmentary and unreliable data in a damage assessment center and development of a war game model for assessing military missile effectiveness. In a separate assignment, he held the position of Manager of Systems Analysis for three years.

5. Interactions

5.1 EIA Symposium

P. B. Moranda presented a briefing on modeling in a session of a symposium in November 1977 at San Diego sponsored by the Electronics Industry Associates. Participated in a 3-day workshop on Software Reliability at the same symposium.

5.2 3rd Int. Conference on Software Engineering

P.B. Moranda participated in the bird-of-feather meeting on software modelling with A. Sukert, RADC, M. Shooman of Polytechnic Institute of New York, B. Littlewood of the City University of London and A. Goel of Syracuse University.

5.3 Z. Jelinski - In October 1977 at the Electronics Industries Associates Eleventh Annual Data and Configuration Management Workshop in San Diego co-chaired panel entitled Software Quality Assurance and Reliability as it relates to Configuration Management. Software Quality Metrics and factors influencing the quality were the main topics of discussions. Recommendations were made to the government and industry.

5.4 Z. Jelinski - In October 1977 at the AIAA/NASA/IEEE/ACM Computers in Aerospace Conference in Los Angeles presented two papers:

1. "Decreasing Design Errors and Problems with Support Software as Deliverables."
2. With K. V. Smith "HOLDET - Higher Order Language Evaluation Tool". In ensuing discussiogs for both papers Software reliability aspects and software quality metrics were discussed.

5.5 Z. Jelinski - In June 1978 at IEEE/NBS/IAS Workshop on "Organizing ADP Projects" was a chairman of a panel for Functional Structure. Advantages and disadvantages of various software management organizations were discussed. A number of case studies were made. The question of software reliability was correlated to the project organization structure.

5.6 Z. Jelinski - In March 1978 at NSIA/SQRAC Workshop on Software Reliability in Arlington, Virginia, chaired a panel on "Software Quality Models and Metrics - discussions included the assessment and application of Software Reliability Models to perspective problems.

6. References

1. D. J. Reifer. A Glossary of Software Tools and Techniques, Computer, July 1977.

2. C. V. Ramamoorthy and S. F. Ho. Testing Large Software with Automated Software Evaluation Systems. IEEE Transactions on Software Engineering March 1975; Vol. SE-1, No. 1.

3. J. Goodenough and S. L. Gerhart. Toward a Theory of Test Data Selection. Proceedings of International Conference on Reliable Software, Los Angeles, California, 21-25 April 1975.

4. W. E. Howden. Methodology for the Automatic Generation of Program Test Data. TR No. 41, McDonnell Douglas, February 1974.

5. B. Elspas, M. W Green, K. N. Levitt, and R. J. Waldinger. Research in Interactive Program Proving Techniques: SRI Report 8398-II, Stanford Research Institute, 1972, Menlo Park, California.

6. J. King. Symbolic Execution and Program Testing. Communications of the ACM, July 1976.

7. L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. IEEE Transactions on Software Engineering, Sept. 1976; SE-2, No. 3.

8. L. G. Stucki, Program Evaluation and Tester: PET. McDonnell Douglas M2085074, 1974.

9. L. G. Stucki. Automatic Generation of Self-Metric Software. Proceedings of the IEEE 1973 Symposium on Computer Software Reliability,New York, 1973.

10. P. B. Moranda. Quantitative Methods for Software Reliability Measurements. McDonnell Douglas Astronautics Company, MDC G6553, Final Report on AFOSR F44620-74-C-008, December 1976.

11. E. I. Cohen and L. J. White. A Finite Domain-Testing Strategy for Computer Program Testing. (CSU-CISRC-TR-77-13). The Ohio State University, Columbus, Ohio, August 1977.

12. Z. Jelinski and P. B. Moranda. Software Reliability Research in Statistical Computer Performance Evaluation. Walter Freiberger, Ed. Academic Press, New York, 1972.

13. P. B. Moranda. Estimation of A Priori Software Reliability. Computer Science and Statistics Interface Symposium, February 1975, Los Angeles, California.

14. W. Miller and D. L. Spooner. Automatic Generation of Floating-Point Test Data. IEEE Transactions on Software Engineering, September 1976, Vol. SE-2, No. 3.

15. A Fitzsimmons and T. Love, "A Review and Evaluation of Software Science", ACM Computing Surveys, Vol.1, No.1, March 1978.

16. J. L. Elshoff. An Investigation into the Effects of the Counting Method Used on Software Science Measurements. IEEETSE, Vol. SE-2, No. 4, December 1976.

17. T. J. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, December 1976; Vol. SE-2, No. 4.

18. T. Gilb, Software Metrics, Winthrop Publishers, Inc., Cambridg, Mass. 1977.

19. G. J. Schick and R. W. Wolverton. An Analysis of Competing Software Reliability Models. IEEETSE, March 1978; Vol. SE-4, No. 2, (reviewed in section 3.1.8).

20. K. Okumoto and A. Goel. A Model for Reliability and Other Quantitative Measures of Software System Subject to Imperfect Debugging (submitted for publication). Summary available in RADC-TR-77-112, March 1977.

21. E. C. Miller. Tutorial on Program Testing Techniques. COMSAC77, Chicago , Illinois, 8-11 November 1977.


79